Project 2: Continuous Control

The main aim of this project is to train an agent to learn by itself to move to target locations. The goal of the agent is to maintain its position at the target location for as many time steps as possible; hence, a reward of +0.1 is provided for each step the agent's hand is in the goal location.

The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector is a number between -1 and 1.

For this project, an environment built in Unity is used.

Learning Algorithm

To solve the environment, a Deep Deterministic Policy Gradients (DDPG) algorithm is used. The DDPG code considers only a single agent, and with each step, the agent adds it experience to the replay buffer, and the actor and critic networks are updated, using a sample from the replay buffer.

Some of the parameters used include:

- `BUFFER_SIZE = int(1e6): This is the replay buffer size`
- `BATCH_SIZE = 128: This is the size of each minibatch`
- `GAMMA = 0.99: This is the discount factor`
- `TAU = 1e-3: This is used for soft update of target parameters`
- `LR_ACTOR = 1e-3: This is the learning rate of the actor`
- `LR_CRITIC = 1e-3: This is the learning rate of the critic`
- `LEARN_EVERY = 20: This is the learning timestep interval`
- `LEARN_NUM   = 10: This is the number of learning passes`
- `GRAD_CLIPPING = 1.0: For gradient clipping`
- `OU_SIGMA = 0.15: Ornstein-Uhlenbeck noise parameter`
- `OU_THETA = 0.05: Ornstein-Uhlenbeck noise parameter`
- `EPSILON       = 1.0: Epsilon in the noise process (act step)`
- `EPSILON_DECAY = 1e-6: Decay rate for epsilon`

Model Architecture

To solve the environment, three fully connected layers is used for the actor-critic network. Furthermore, a batch normalization layer is also used.

For the actor network:

```
self.fc1 = nn.Linear(state_size, fc1_units)

self.bn1 = nn.BatchNorm1d(fc1_units)

self.fc2 = nn.Linear(fc1_units, fc2_units)

self.fc3 = nn.Linear(fc2_units, action_size)

x = F.relu(self.bn1(self.fc1(state)))
```

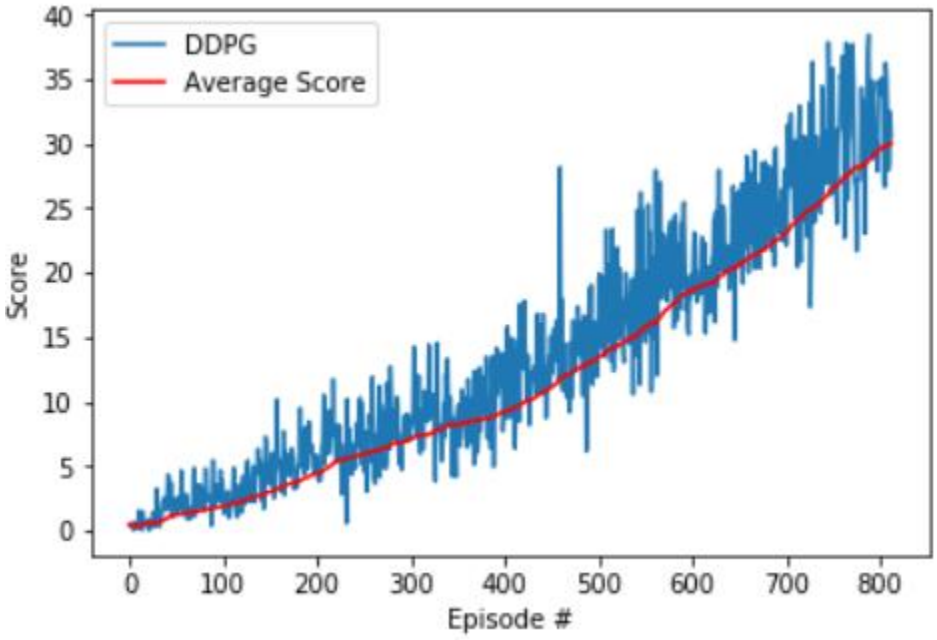
```
x = F.relu(self.fc2(x))
return torch.tanh(self.fc3(x))
```

For the critic network:

```
self.fcs1 = nn.Linear(state_size, fcs1_units)
self.bn1 = nn.BatchNorm1d(fcs1_units)
self.fc2 = nn.Linear(fcs1_units+action_size, fc2_units)
self.fc3 = nn.Linear(fc2_units, 1)
xs = F.relu(self.bn1(self.fcs1(state)))
x = torch.cat((xs, action), dim=1)
x = F.relu(self.fc2(x))
return self.fc3(x)
```

Result

The agent is trained until the environment is solved. To achieve this, the agent obtained an average score of at least +30 over the last 100 episodes.



The environment was solved in 713 episodes with an average score of 30.07.

Ideas for Future Work

For future work, I intend look at algorithms like PPO, A3C, and D4PG that uses multiple (non-interacting , parallel) copies of the same agent to distribute the task of gathering experience. This will be the best for solving the second version of the Unity environment which contains 20 identical agents.